Nonblocking Epochs in MPI One-Sided Communication

Judicael A. Zounmevo,* Xin Zhao,† Pavan Balaji,‡ William Gropp,† and Ahmad Afsahi*

*Queen's University, Kingston, ON, {judicael.zounmevo, ahmad.afsahi}@queensu.ca
†University of Illinois at Urbana-Champaign, {xinzhao3, wgropp}@illinois.edu
‡Argonne National Laboratory, balaji@mcs.anl.gov

***Abstract*—The synchronization model of the MPI one-sided communication paradigm can lead to serialization and latency propagation. For instance, a process can propagate non RMA communication-related latencies to remote peers waiting in their respective epoch-closing routines in matching epochs. In this work, we discuss six latency issues that were documented for MPI-2.0 and show how they evolved in MPI-3.0. Then, we propose entirely nonblocking RMA synchronizations that allow processes to avoid waiting even in epoch-closing routines. The proposal provides contention avoidance in communication patterns that require back-to-back RMA epochs. It also fixes the latency propagation issues. Moreover, it allows the MPI progress engine to orchestrate aggressive schedulings to cut down the overall completion time of sets of epochs without introducing memory consistency hazards. Our test results show noticeable performance improvements for a lower-upper matrix decomposition as well as an application pattern that performs massive atomic updates.**

***Keywords*—MPI, one-sided, RMA, nonblocking synchronizations, latency propagation**

I. Introduction

In the Message Passing Interface (MPI) [1], one-sided communications, also called remote memory access (RMA), must happen inside critical section-like regions called epochs. An epoch is started by one of a set of RMA synchronization calls and ended by a matching synchronization call. MPI-2.0 RMA was criticized for its synchronization burden [2] and various constraints that make it difficult to use in certain situations. In that regard, MPI-3.0 represents an undeniable and important improvement because it alleviates quite a few of the aforementioned constraints and introduces many new features for avoiding frequent synchronization invocations. For instance, the introduction of the request-based one-sided communications in MPI-3.0 RMA eliminates the need for the communication initiator to always make a synchronization call in order to detect RMA completion at the application level. However, these improvements still leave out scenarios where the synchronization burden cannot be avoided. For instance, the request-based RMA communication calls are reserved for only a certain category of epochs, called *passive target*. Even in passive target epochs, the need arises sometimes to isolate different communications in distinct epochs, for instance to guarantee atomicity. Serialization thus becomes unavoidable because epoch-ending routines are blocking. The possibly blocking nature of MPI one-sided communication synchronizations is actually the cause of six kinds of latency issues first documented in [3] as MPI-2.0 RMA inefficiency patterns. Four of these six inefficiency patterns were difficult to avoid or work around. These latency issues are the consequence of late peers propagating latencies to remote processes blocking on RMA synchronizations.

In this work we address the synchronization burden by making the one-sided communications nonblocking from start to finish, if needed. Because the entire MPI-RMA epoch can be nonblocking, MPI processes can issue the communications and move on immediately. Conditions are thus created for (1) enhanced communication/computation overlapping, (2) enhanced communication/communication overlapping, and (3) delay propagation avoidance or mitigation via communication/delay overlapping. Our proposal solves all four inefficiency patterns, plus a fifth one introduced and documented in this work. Additionally, the proposal allows various kinds of aggressive communication scheduling meant to reduce the overall completion time of multiple epochs initiated from the same process. Our test results show up to 39% performance improvement for a transactional application kernel and up to 50% improvement for a lower-upper matrix decomposition RMA implementation. To the best of our knowledge, this work represents the first attempt to remove all wait phases from the lifetime of MPI RMA epochs.

The rest of the paper is organized as follows. Section II discusses background material. Section III introduces the inefficiency patterns. Section IV shows the impact of the work. Section V presents the nonblocking synchronization API. Section VI describes the semantics and behaviors to expect from nonblocking synchronizations and epochs. Section VII discusses a few key design choices. Section VIII presents the experimental results. Section IX discusses some related work, and Section X summarizes our conclusions and briefly mentions future work.

II. Background

One-sided communications occur over a *window* object that defines (1) the memory regions that each process intends to expose for remote accesses and (2) a communication scope encompassing a set of processes. In one-sided communications, the *origin* process specifies all the communication parameters while the *target* process remains passive. MPI defines two classes of RMA synchronizations: *active target*, where the target explicitly opens an epoch as well, and *passive target*, where the target does not make epoch calls. Active target epochs can be fence-based, in which case they are created and ended by `MPI_WIN_FENCE`. Active target epochs can also

be based on general active target synchronization (GATS), in which case origin-side *access* epochs are opened and closed by MPI_WIN_START and MPI_WIN_COMPLETE, respectively, and target-side *exposure* epochs are opened and closed by MPI_WIN_POST and MPI_WIN_WAIT, respectively. GATS is a fine-grained style of *active target* compared with fence. In passive target, the origin requests a lock from the target. The request can be from a single target, in which case the underlying epoch is opened and closed with MPI_WIN_LOCK and MPI_WIN_UNLOCK, respectively. MPI_WIN_LOCK can make exclusive (MPI_LOCK_EXCLUSIVE) or shared (MPI_LOCK_SHARED) lock requests. The origin in passive target can also target all the processes in the RMA window for a shared lock with an epoch opened and closed with MPI_WIN_LOCK_ALL and MPI_WIN_UNLOCK_ALL, respectively. RMA communications occurring inside a passive target epoch can be completed without closing the epoch by using one of a set of ***flush*** routines.

III. The Burden of Blocking RMA Synchronizations: Inefficiency Patterns

The MPI one-sided inefficiency patterns [3], [4] are situations that force some unproductive wait or idleness on a peer involved implicitly or explicitly in RMA communications. The inefficiency patterns are a consequence of the blocking nature of RMA synchronization routines. They are listed as follows:

- *Late Post*: A GATS-related inefficiency where MPI_WIN_COMPLETE or MPI_WIN_START must block because the target is yet to issue MPI_WIN_POST.
- *Early Transfer*: A pattern that occurs when an RMA communication call blocks because the target epoch is not yet exposed.
- *Early Wait*: A pattern that occurs when MPI_WIN_WAIT is called while the RMA transfers are not completed yet.
- *Late Complete*: The delay between the end of the last RMA transfer and the actual invocation of MPI_WIN_COMPLETE. That delay propagates to the target as an unproductive wait. The target-side MPI_WIN_WAIT must block as long as MPI_WIN_COMPLETE is not invoked by the origin; and when that invocation is delayed for reasons other than RMA transfers, Late Complete occurs.
- *Early Fence*: The wait time associated with an epoch-closing fence call that occurs before the RMA transfers complete. Early Fence is the fence epoch equivalent of Early Wait for GATS epochs.
- *Wait at Fence*: A superset of Early Fence. A closing fence call in any process must block until the same call occurs in all the processes in the group over which the RMA window is defined. If any process delays its call to fence beyond the end of its last RMA transfer, then it inflicts a Wait at Fence inefficiency to the other processes that issue their epoch-closing fence earlier for the same RMA window.

We have also identified another inefficiency pattern not documented in [3], [4]. Since a passive target epoch does not involve an explicit epoch from the target side, situations of inefficient latency transfers are less obvious. When at least two origins are considered, however, we can define a new inefficiency pattern inflicted by current lock holders to subsequent lock requesters. That inefficiency pattern, which we call ***LateUnlock***, can occur under two conditions:

1) The current holder possesses the lock exclusively. All the RMA transfers of the epoch have already completed, but the holder delays the call to MPI_WIN_UNLOCK while there are any number of requesters willing to acquire the same lock (exclusively or not).
2) The current holders possess the lock in a shared fashion. All the RMA transfers are completed for all the holders at a time t. At least one holder is holding the lock beyond t while an exclusive lock requester is waiting.

Which inefficiency pattern can occur might depend on the MPI implementation the application is using. Actually, the MPI standard does not specify the blocking or nonblocking nature of most epoch-opening synchronizations. The specification simply requires that RMA operations avoid accessing nonexposed remote epochs and that epoch-closing routines not exit until all the RMA transfers originating from or directed towards the epoch are done transferring; at least locally. In practice however, most modern and major MPI libraries [5], [6], [7] provide nonblocking epoch-opening routines—and rightfully so, because the blocking design of those phases of one-sided communication is documented as suboptimal [8], [9]. Consequently, most MPI libraries avoid incurring Late Post on MPI_WIN_START invocation. Late Post can still be incurred at MPI_WIN_COMPLETE, however.

Even though MPI-2.2 did not strictly specify RMA communication calls as nonblocking, implementations typically provide only nonblocking versions of those routines, for the same reasons described in [8], [9]. Thus, Early Transfer is generally avoided altogether. Additionally, MPI-3.0 has now explicitly specified the RMA communication calls as nonblocking, making the Early Transfer pattern nonexistent as per the standard itself. Furthermore, Early Wait can be mitigated by using MPI_WIN_TEST. Late Post, Late Complete, Early Fence, and Wait at Fence, however, are still as much of a burden in MPI-3.0 RMA as they were in MPI-2.0 RMA.

IV. Impacts of Nonblocking RMA Epochs

The generic nonblocking handling mechanism that exists in MPI operates in two phases: *Initiation* and *Completion*. The initiation (e.g., MPI_ISEND) is always nonblocking and corresponds to the moment the operation is issued. The initiation returns a REQUEST object that is used later to detect completion with any of the ***wait*** or ***test*** family of MPI functions. Our proposal of nonblocking synchronizations is conceived to operate the same way. The initiation is provided with the new API presented in Section V; and completion is based on the already existing ***wait*** or ***test*** family of functions. The benefits of nonblocking synchronizations are described in the following subsections.

A. Opportunistic Message Progression

Every time it gets a chance to use the CPU for a communication c_i for instance, the progress engine *opportunistically* tries to trigger or make progress on (if possible) all previously pending communications c_{i-k} that could not be triggered earlier because of some unsatisfied precondition. Opportunistic message progression is beneficial only to communications that are already issued and pending inside the MPI middleware. Nonblocking synchronizations increases the effectiveness of opportunistic message progression for RMA epochs by allowing several of them, no matter their kinds (passive or active target) to reside in pending states inside the progress engine.

B. Contention Avoidance

Two epochs E_k and E_{k+1} posted back-to-back lead to E_{k+1} potentially suffering contention because of the closing synchronization of E_k. Such a situation can occur in algorithms that perform massive transactions in potentially unstructured ways. The communication pattern goes as follows. At any given time, a set of peers $\{P_i\}$ can update another (not necessarily disjoint) set $\{P_j\}$ of processes. Processes do not know ahead of time how many updates they will get; nor can they determine where these updates will originate from or what buffer offset they will modify. Consequently, the updates are best fulfilled one-sidedly by the updating peers. Each update is atomic and is best fulfilled inside exclusive lock epochs. This communication pattern suffers contention with blocking synchronizations because each update must wait for the previous one to complete. With nonblocking synchronizations, multiple epochs (or updates) can be pending simultaneously, and some could even complete out of order, leading to an increased transaction throughput.

C. Fixing the Inefficiency Patterns

We mentioned in Section III that the Late Post, Late Complete, Early fence, Wait at Fence and Late Unlock inefficiency patterns are yet to be fixed. All five inefficiency patterns find a solution with nonblocking epochs.

1) Late Post: We define t_0 as the invocation time of the origin-side epoch-closing routine. We assume that `MPI_WIN_POST` is late and occurs D_p after t_0. The data transfer duration of the RMA communications in the epoch is D_{tr}; and the time needed to close the origin-side epoch if all transfers are already completed is ε. In these conditions, the earliest the next origin-side activity can start after `MPI_WIN_COMPLETE` is

$$t_{1\_blocking\_epoch} = t_0 + D_p + D_{tr} + \varepsilon. \quad (1)$$

With a nonblocking origin-side epoch closing, the RMA transfer duration does not propagate to the next activity. More important, any delay created by the target not exposing its epoch on time does not propagate to the next origin-side activity. Consequently, the earliest the next activity can start becomes

$$t_{1\_nonblocking\_epoch} = t_0 + \varepsilon. \quad (2)$$

Equation 2 shows that nonblocking epochs allow the origin process to completely mitigate the Late Post situation if the next activity does not depend on data produced in the epoch. Then, in absence of data dependency, since the next activity starts sooner than $t_{1\_blocking\_epoch}$, it overlaps (partially or entirely) with the delay and RMA transfer of the previous epoch even if it occurs after the epoch is over, allowing the overall completion of both activities to be reduced.

2) Early Fence: We define t_0 here as the time when `MPI_WIN_FENCE` or its nonblocking equivalent is invoked. With the previous definitions for all the other relevant time variables, the earliest moment the next activity can start with a blocking fence is

$$t_{1\_blocking\_epoch} = t_0 + D_{tr} + \varepsilon. \quad (3)$$

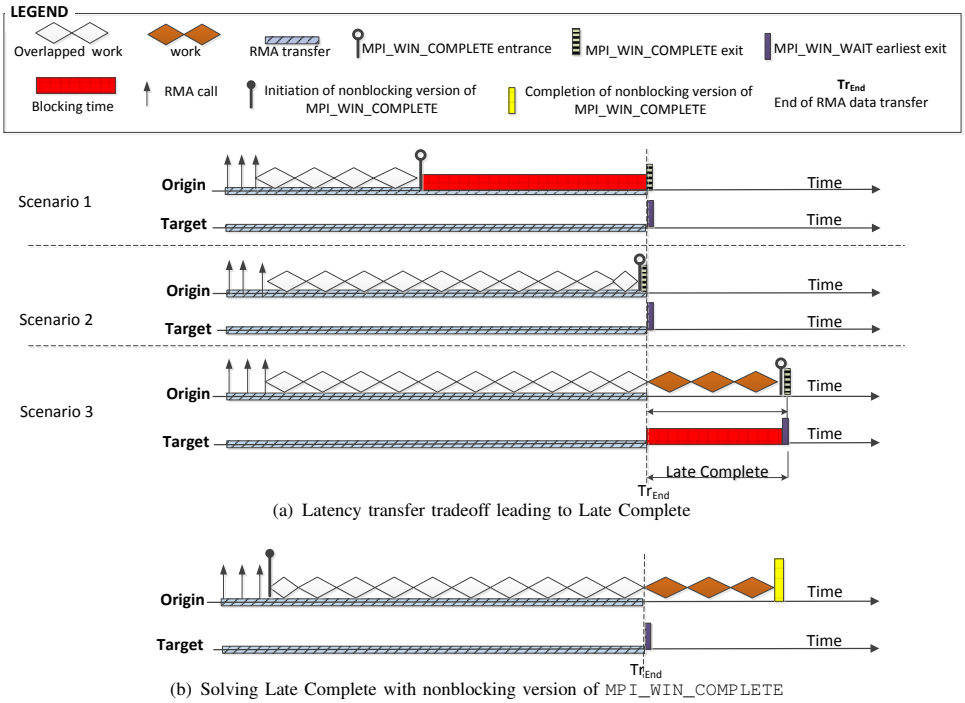
With a nonblocking version of `MPI_WIN_FENCE`, the next activity can also start at $t_{1\_nonblocking\_epoch}$, as expressed by Equation 2, with the same positive consequences mentioned previously.

3) Late Complete: Blocking epochs offer no indication as to the right synchronization call timing for maximizing performance. Actually, they offer conflicting strategies for latency avoidance or mitigation. Epochs are critical section-like regions; and, as such, they should be kept as short as possible (scenario 1 of Fig. 1(a)). At the same time, CPU idling should be avoided. Therefore, useful work should be overlapped with the communication of an epoch if that communication is thought to have a somehow lasting transfer time (scenario 3 of Fig. 1(a)). The performance-savvy MPI programmer resorts to this second approach to avoid the CPU idleness of scenario 1 in Fig. 1(a). It is unrealistic to expect the occurrence of scenario 2 of Fig. 1(a) because the application cannot calibrate its work length to be exactly the length of its data transfer.

Because these epochs are like critical sections, `MPI_WIN_COMPLETE` is invoked as quickly as possible. For an access epoch, however, there is no guarantee that the corresponding exposure epoch will be opened on time. In a GATS setting, this means that an early `MPI_WIN_COMPLETE` call increases both the risk and magnitude of Late Post suffering by origin processes.

The Late Complete inefficiency results from the hunt for communication/computation overlapping and could therefore be the consequence of applying recommended HPC programming practices. The situation leading to Late Complete (scenario 3 in Fig. 1(a)) is also a selfish attempt made by the origin process to avoid stalling while its own RMA transfers are in progress. By doing so, the origin process is better off; but it potentially transfers an unjustified wait time to the target. The alternative is that shown in scenario 1 of Fig. 1(a), which guarantees the absence of the Late Complete inefficiency pattern at the expense of the origin process. The two (realistic) scenarios 1 and 3 are therefore the two aspects of an unavoidable tradeoff where one peer potentially suffers some undesirable wait. With a nonblocking version of `MPI_WIN_COMPLETE` (Fig. 1(b)), the tradeoff disappears because it becomes possible to keep the origin in scenario 3 while simultaneously having the target in scenario 1 of Fig. 1(a).

4) Wait at Fence as the Risky Remedy for Early Fence: Early Fence and Wait at Fence are the two unfortunate options of a blind decision-making strategy. In order to avoid Early Fence with blocking epoch routines, a process should issue its fence call later. Unfortunately, by doing so, it might inflict Wait at Fence to the other participant processes. Once again, nonblocking epoch routines allow every participant to be selfish without inflicting any inefficiency to the other participants.



(a) Latency transfer tradeoff leading to Late Complete

(b) Solving Late Complete with nonblocking version of `MPI_WIN_COMPLETE`

Fig. 1: Impact of nonblocking epochs on Late Complete

With the right design in place, the nonblocking fences are issued early for every participant; the subsequent calls to ***test*** or ***wait*** can be delayed as much as possible without any latency transfer to remote peers.

5) Late Unlock: Late Unlock can be analyzed by replacing the "origin" in Fig. 1 with the "current lock holder" and the "target" with the "subsequent lock requester." There is no incentive for the current lock holder to issue `MPI_WIN_UNLOCK` early; as it might experience the same stalling as the origin in scenario 1 of Fig. 1(a). Unfortunately, by putting itself in scenario 3 of Fig. 1(a), the current lock holder could inflict an undesirable wait time to a subsequent lock requester. Just as in the case of Late Complete, a nonblocking version of `MPI_WIN_UNLOCK` completely voids the tradeoff.

V. Nonblocking API

For each potentially blocking epoch function `MPI_WIN_FUNC(LIST_OF_PARAM)`, a nonblocking version of the form `MPI_WIN_IFUNC(LIST_OF_PARAM,REQUEST)` is provided. `LIST_OF_PARAM` is the list of parameters of the blocking version; `REQUEST` is an output parameter used to detect completion with a function from the ***test*** or wait family of MPI routines.

The nonblocking epoch-opening API is composed of `MPI_WIN_IPOST`, `MPI_WIN_ISTART`, `MPI_WIN_IFENCE`, `MPI_WIN_ILOCK`, and `MPI_WIN_ILOCK_ALL`. While modern MPI libraries tend to provide nonblocking epoch-opening routines [8], [5], [6], [7], [9], the MPI standard is not specific about the blocking or nonblocking nature of all these functions, meaning that their behavior is implementation-dependent. The API provided in this section specifies the uniform ambiguity-free nonblocking nature of its functions. `MPI_WIN_IPOST` is provided solely for uniformity and completeness; as it was already specified as nonblocking in MPI-3.0.

The nonblocking epoch-closing API is composed of `MPI_WIN_IWAIT`, `MPI_WIN_ICOMPLETE`, `MPI_WIN_IFENCE`, `MPI_WIN_IUNLOCK`, and `MPI_WIN_IUNLOCK_ALL`. Recall that `MPI_WIN_FENCE` is used for both epoch opening and epoch closing and that MPI-3.0 already provides `MPI_WIN_TEST` as the nonblocking equivalent of `MPI_WIN_WAIT`uf. However, the new `MPI_WIN_IWAIT` that we propose remains relevant: in fact, compared with `MPI_WIN_TEST`, the combination of `MPI_WIN_IWAIT` with the ***test*** family of nonblocking handling is more powerful because it allows the asynchronous and wait-free initiation of subsequent epochs. `MPI_WIN_TEST` detects the completion of the current exposure epoch in a nonblocking manner; but since no other exposure epoch can be opened until the completion actually occurs, it does not prevent application-level epoch serialization. It simply prevents the CPU from idling while waiting for the currently active exposure epoch to complete.

The nonblocking ***flush*** API is composed of `MPI_WIN_IFLUSH`, `MPI_WIN_IFLUSH_LOCAL`, `MPI_WIN_IFLUSH_ALL` and `MPI_WIN_IFLUSH_LOCAL_ALL`.

VI. Semantics, Correctness, and Progress Engine Behaviors

For an epoch, we distinguish between *application-level lifetime* and *internal lifetime*. We use the terms "open" and "closed" to define the boundaries of the application-level lifetime of an epoch. The internal lifetime takes place inside the middleware: it starts when the epoch is internally *activated* for progression by the progress engine, and it ends when the epoch is done being progressed and all the internal completion notifications have been sent to all the relevant peers. The terms "activated" and "completed" define the boundaries of

the internal lifetime of an epoch; and an epoch is said to be *active* inside those boundaries. We designate by *deferred epoch* an epoch that cannot be activated as soon as it is opened at application level. A deferred epoch is different from an epoch that is not yet granted access. In fact, all kinds of epochs can be deferred, including exposure epochs for which access acquisition is not meaningful.

A. Semantics and Correctness Clarifications

The statements in this section do not alter any aspect of the MPI-3.0 specification except for the meaning of epoch closing. The following rules, meant to define constraints, clarifications, and recommendations, apply:

1) An epoch can be opened in a nonblocking fashion and closed in a blocking fashion and vice versa. In fact, any combination of blocking and nonblocking routines can be used for the synchronization routines that make an epoch.
2) An epoch **closed** with a nonblocking synchronization might not **complete** until completion is explicitly detected with one of the ***test*** or ***wait*** family of functions. Accessing the buffers touched by such an epoch remains unsafe until completion is detected at application level.
3) In active target, access and exposure epochs match one another in a FIFO manner. The oldest deferred access epoch is always the next to match the oldest deferred exposure epoch, and vice versa. If any of the sides does not have deferred epochs, then the match occurs between the oldest deferred epoch on one side and the next opened epoch on the other side.
4) For a given process, epochs are always activated serially in the order where they become pending inside the progress engine. Consequently, if the epoch E_k cannot be activated yet, then E_{k+1} cannot be activated either. This rule means that epochs are not skipped. This rule **strictly** means "E_{k+1} is activated after E_k is activated," not "E_{k+1} is activated after E_k completes" (see Section VI-B).
5) `MPI_WIN_IFENCE` entails a barrier semantics wherever `MPI_WIN_FENCE` does, that is, whenever the fence call ends an epoch. In particular, if a call to `MPI_WIN_IFENCE` must close epoch E_k and open epoch E_{k+1}, the progress engine of each process must internally delay activating E_{k+1} until it receives the E_k completion notifications from all the other peers encompassed by the RMA window. The delay impacts how long it takes for completion to be fulfilled for `MPI_WIN_IFENCE` but it must produce no noticeable blocking call at the application level.

B. Info Object Key-Value Pairs and Aggressive Progression Rules

To guarantee correctness by default, the progress engine does not activate an epoch while another one is still active; that is, by default "E_{k+1} is activated after E_k completes." This default behavior does not defeat the purpose of nonblocking epochs or synchronizations, since opportunistic message progression is still beneficial for all the epochs serialized inside the progress engine; these epochs are progressed as early as possible when they are already pending inside the middleware.

Further optimizations are possible beyond the sole opportunistic message progression advantage, especially when the programmer possesses certain guarantees with respect to consistency hazards. Thus, in addition to the API, we provide the following info object, key-controlled Boolean flags that the programmer can associate with an RMA window:

- `MPI_WIN_ACCESS_AFTER_ACCESS_REORDER` (`A_A_A_R`): If its value is 1, then the progress engine can activate and progress any origin-side epoch even if an immediately preceding origin-side epoch is still active.
- `MPI_WIN_ACCESS_AFTER_EXPOSURE_REORDER` (`A_A_E_R`): If its value is 1, then the progress engine can activate and progress any origin-side epoch even if an immediately preceding exposure epoch is still active.
- `MPI_WIN_EXPOSURE_AFTER_EXPOSURE_REORDER` (`E_A_E_R`): If its value is 1, then the progress engine can activate and progress any target-side epoch even if an immediately preceding target-side epoch is still active.
- `MPI_WIN_EXPOSURE_AFTER_ACCESS_REORDER` (`E_A_A_R`): If its value is 1, then the progress engine can activate and progress any target-side epoch even if an immediately preceding origin-side epoch is still active.

If any of these four flags is enabled, the RMA communications of a subsequent epoch E_{k+1} can end up being transferred before those of a previous epoch E_k. If the epochs contain any `MPI_GET`, `MPI_RGET`, `MPI_GET_ACCUMULATE`, `MPI_RGET_ACCUMULATE`, `MPI_FETCH_AND_OP`, or `MPI_COMPARE_AND_SWAP`, write reordering can occur in the origin address space with respect to the chronology of E_k and E_{k+1}. Similarly, if the epochs contain `MPI_PUT`, `MPI_RPUT`, `MPI_ACCUMULATE`, `MPI_RACCUMULATE`, `MPI_GET_ACCUMULATE`, `MPI_RGET_ACCUMULATE`, `MPI_FETCH_AND_OP` or `MPI_COMPARE_AND_SWAP`, write reordering can occur in some targets with respect to the chronology of E_k and E_{k+1}. Write reordering is not a desirable outcome because it bears hazards. Justifiably, all these flags are disabled by default. It is assumed that the HPC programmer who activates these flags can guarantee, using the knowledge of the data access pattern of the application, that the RMA activities of concurrently progressed epochs involve strictly disjoint memory regions. These flags operate at the window level and independently for each window. They allow the progress engine to perform aggressive message progression by completing epochs out of order if required.

The optimization flags do not apply to any two adjacent epochs of which at least one is opened by `MPI_WIN_LOCK_ALL`, `MPI_WIN_FENCE`, or their respective nonblocking equivalents. For any two adjacent epochs of which one is based on `MPI_WIN_LOCK_ALL` or `MPI_WIN_ILOCK_ALL`, if the other epoch is based on `MPI_LOCK_SHARED`, then recursive locking can occur for a certain peer if the optimization flags were to take effect. If the other epoch is not based on `MPI_LOCK_SHARED`, then the flags create the risk of violating, in at least one process, the MPI standard constraint that disallows an RMA window to be simultaneously locked and

exposed. As for MPI_WIN_FENCE and MPI_WIN_IFENCE, they simultaneously open an access and exposure epoch on every process in the RMA window. They additionally entail a barrier semantics every time they close an epoch. Because of the resulting complexity, enabling the optimization flags on fence epochs calls for further analysis, which we leave for future efforts.

VII. Design and Realization Notes

We discuss here a design that we implemented in MVAPICH. The need arose to (1) completely decouple the exit of synchronization routines from the preceding RMA communication calls, (2) manage arbitrary numbers and combinations of simultaneously pending epochs per RMA window, and (3) implement new concepts that cannot easily be patched onto the existing MVAPICH RMA implementation. Thus, we opted for a complete redesign that covers both blocking and nonblocking synchronizations as well as the RMA communication calls. The implementation is done over InfiniBand with OFED verbs [10]. This section presents some of the many design decisions.

A. Deferred Epochs and Epoch Recording

Deferred epochs are middleware-level concepts that we introduced in Section VI. An epoch is deferred when its immediate activation would violate some rule. A deferred epoch is *recorded* until it is closed at application level. When it becomes activated later, it is *replayed* internally. Certain aspects of deferred epochs are recorded by saving exactly function call arguments and reissuing internal versions of the calls upon activation. Other aspects are saved after a certain amount of processing. An epoch can remain deferred even until it is closed at the application level, in which case it is internally flagged as closed. Deferred epochs are hosted in a deferred epoch queue attached to their RMA windows. Every time an active epoch is completed internally, the RMA progress engine scans all the existing deferred epochs and activates in sequence all those that do not violate any rule. The scan stops when the first deferred epoch is encountered that fails activation conditions.

B. Epoch Ids and Epoch Matching

We define T_i as some group of processes all acting as targets in the same RMA window. Let us consider three processes P_0, P_1, and P_2. P_1 is defined to belong to the groups T_0, T_1, T_2, T_3, and T_5. P_2 belongs to T_4 and T_5. P_0 is an origin process that opens six access epochs successively toward T_0 to T_5 in order. The 6^{th} access epoch of P_0 is the 5^{th} toward P_1 and the 2^{nd} toward P_2. Because of the nonblocking synchronizations, P_2 for instance could open its 2^{nd} exposure far ahead of P_0 opening its overall 6^{th} access epoch to match that second exposure of P_2. This example shows that when a target grants access to an origin that is several epochs late, the granted access notification must persist for the origin in order to see it when it catches up. Since there is a need for keeping a history of granted accesses, nonblocking epochs create a problem reminiscent of message queue processing [11]. Nevertheless, we managed to fulfill the epoch matching in $O(1)$ for both running time and space cost. In fact, for a given RMA window, a single triple of 64-bit numbers is required to manage the epoch-matching history between a remote process P_r of rank r and the local process P_l, without respect to the number of pending epochs that link P_r and P_l. We emphasize that P_l and P_r can be the same process if $l == r$. In each local process P_l and for each remote process P_r, we define a triple $\omega_r = \langle a_l, e_l, g_r \rangle$ which is made of the number (so far) of (1) accesses requested from P_l to P_r, (2) exposures opened from P_l to P_r, and (3) accesses obtained from P_r by P_l, respectively. We note that g_r is updated one-sidedly, via RDMA or shared memory, by P_r, while a_l and e_l are updated locally by P_l. Only activated epochs modify ω_r. Deferred epochs can nevertheless modify other counters, for instance to prevent recursive shared locking when multiple activation occurs. Even though granting a passive target lock does not create an exposure epoch, the host process of a lock still updates e_l locally and g_r remotely in the process it is granting the lock to. The *access Id* of a new access epoch E_i toward P_r is $A_i = ++a_l$. Then $A_i > g_r$ means that P_r is at least 1 exposure epoch late compared with E_i in P_l. If $A_i \leq g_r$, then P_r has already granted the access required by E_i as well as all the k subsequent accesses (for $k = g_r - A_i$). $A_i \leq g_r$ thus means that E_i can perform its RMA communications toward P_r. To close an access epoch toward a given target, P_l sends a done packet containing A_i to match the *exposure Id* of the matching epoch in the target. Closing a lock epoch requires a different kind of done packet.

VIII. Evaluation

We describe here comparison tests between the one-sided communication model provided by MVAPICH 2-1.9 and the new implementation discussed in Section VII. We use the terms "MVAPICH," "New," and "New nonblocking" for the three test series whose results are obtained with the vanilla MVAPICH RMA, the new design with blocking synchronizations, and the new design with nonblocking synchronizations, respectively. The experimental setup is a 310-node cluster. Each node has two Nehalem 2.6 GHz Pentium Xeon CPUs with hyperthreading disabled, 36 GB of memory, and Mellanox ConnectX QDR InfiniBand HCA. The results of each microbenchmark test are the average over 100 iterations. For the application pattern and application tests, each result is the average over 3 iterations.

A. Microbenchmark

Both the blocking and nonblocking synchronization versions of the proof-of-concept implementation offer communication/computation overlapping and RMA latency **at least** on par with the vanilla MVAPICH. Because of space limitations, however, microbenchmark results are shown only for the conceptual improvements brought by the nonblocking synchronizations. The usual generic latency and overlapping observations (with no delay propagation and no late peer arival) can nevertheless be briefly summarized as follows. Both the blocking and nonblocking versions of the new implementation have similar latency performance compared with MVAPICH for all kinds of epochs. MPI_ACCUMULATE with large payloads (more than 8 KB on our test system) do not provide overlapping in any of the implementations because of the need for an internal rendezvous for target-side intermediate buffer to receive the origin-side operand. In all other cases, the new implementation provides full communication/computation overlapping in lock epochs, while MVAPICH provides none

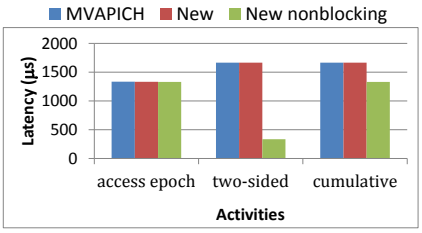


Fig. 2: Mitigating the Late Post inefficiency pattern: observing delay propagation in an origin process

because of its *lazy* lock acquisition approach [12]. In lazy lock acquisition, the locking attempt, and consequently the whole epoch, is not internally fulfilled until `MPI_WIN_UNLOCK` is invoked at application level. For fence epochs, the new implementation provides slightly better overlapping as well; and for GATS, the results are similar for all implementations.

1) Inefficiency Patterns: We discuss here our approach to mitigating the five inefficiency patterns identified earlier. **Late Post:** The test setting is made of a target process P_0 which is $1000\mu s$ late in opening its exposure epoch and a process P_1 meant for two-sided communication. An origin process P_2 first opens an access epoch toward P_0, After the epoch is closed, the process then performs a single two-sided communication of 1 MB with P_1. A single put of 1 MB is performed inside the epoch. The size of the put does not matter. Figure 2 shows the duration till completion of the access epoch, the subsequent activity, and both activities for the origin (cumulative). All measurements are made in P_2 and with a time origin taken at 0. As a reference, in pure latency experimentations, any epoch hosting an `MPI_PUT` of 1MB takes about $340\mu s$ for all three test series. The delay of the Late Post cannot be avoided by the origin-side epoch, as shown by the access epoch length being about $1340\mu s$ for all three test series. While the subsequent activity takes about $1660\mu s$ for the two blocking test series, however, it takes only about $340\mu s$ for the nonblocking test series, proving that the nonblocking synchronization prevented the delay from being propagating beyond the only concerned epoch. In fact, the nonblocking test series overlaps the subsequent activity with the delay of the late post, leading the cumulative latency of all activities to be only the overall latency of the first activity.

Late Complete: The test setting is made of a single origin and a single target. The origin issues a single `MPI_PUT` and then overlaps $1000\mu s$ of work before the blocking call that completes the epoch. In order to make the delay obvious, the work length is purposely chosen to be bigger than $340\mu s$, which is the approximate latency of transferring 1MB of data. As a reminder, epoch closing (e.g., `MPI_WIN_ICOMPLETE`) is distinct from epoch completion (***wait*** or ***test***) with nonblocking synchronizations. With blocking synchronizations, epoch closing and completion are a single routine. This test is performed for multiple message sizes. Figure 3 shows the length of the target-size epoch. The origin-side epoch, which is not shown, achieves communication/computation overlapping for all three test series; that is, the access epoch lasted $max(1000\mu s, RMA\_latency)$. The two blocking test series propagates the totality of the origin-side epoch length to the target; including the delay $d = 1000\mu s - RMA\_latency$. In comparison, the nonblocking test series guarantees that the target waits only for the duration of the actual RMA transfers.

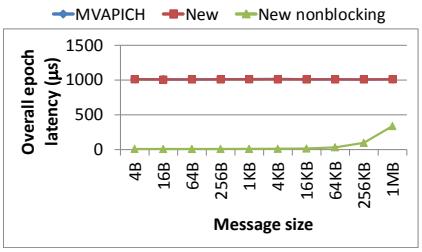


Fig. 3: Mitigating the Late Complete inefficiency pattern: observing delay propagation in a target process

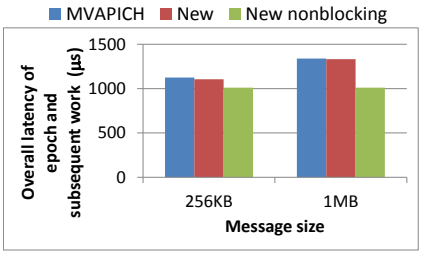


Fig. 4: Mitigating the Early Fence inefficiency pattern: observing communication latency propagation in a target process

Early Fence: The test setting is made of two processes sharing a fence epoch. One of the processes acts like the origin and issues an `MPI_PUT` of either 256 KB or 1 MB. The other process acts like a target. Unlike the Late Post inefficiency, the Early Fence inefficiency, by definition, is not the result of any delay. However, since the wait created by an early epoch-closing fence call corresponds to an idling CPU core (assuming an autonomously progressing network device), the Early Fence situation is still inefficient from an HPC point of view and should therefore be mitigated, if possible. We achieve the mitigation by performing a subsequent CPU-bound activity of $1000\mu s$ after the epoch of the process acting as the target; and the measurements are performed in that process. Figure 4 shows the cumulative latency of both activities for both message sizes. In the nonblocking test series, the subsequent activity is overlapped with the data transfer time of the epoch, even if the epoch was already closed, leading to a cumulative latency of $1010\mu s$. In comparison, both activities are serialized in the blocking test series, leading to much larger cumulative latencies. One can see that an attempt by the blocking test series to overlap the CPU-bound activity **inside** the epoch bears the risk of creating the Wait at Fence inefficiency (see Section IV-C4) and is therefore not equivalent to the safe overlapping achieved by the nonblocking test series in the current experiment.

Wait at Fence: Wait at Fence is the fence equivalent of the Late Complete issue for GATS epochs and has a similar test setting. A single process acts like an origin, while the second one acts like a target in a fence epoch. The measurements are performed from the process acting as target. Figure 5 shows that, unlike the blocking test series, the nonblocking test series prevents the propagation from origin to target of the non RMA-related latency.

Late Unlock: This test requires two origin processes O_0 and O_1 and a single target T. Both O_0 and O_1 lock T exclusively, but we ensure that O_0 issues its lock before O_1 does. Each process issues a single PUT of 1 MB; but O_0 works for $1000\mu s$ before unlocking T. Figure 6 shows the

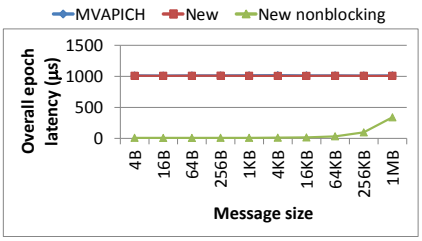


Fig. 5: Mitigating the Wait at Fence inefficiency pattern: observing delay propagation in a target process

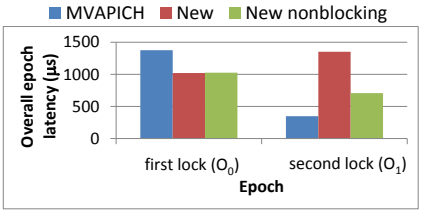


Fig. 6: Mitigating the Late Unlock inefficiency pattern: observing delay propagation to a subsequent lock requester

duration till completion of the first lock epoch (issued by O_0) and the second one (issued by O_1). In MVAPICH, thanks to lazy lock acquisition, even though O_0 requested the lock first at application level, the lock was still actually available by the time O_1 issues both its `MPI_WIN_LOCK` and its `MPI_WIN_UNLOCK`. As a result, O_1 managed to get the lock as soon as it internally asked for it (`MPI_WIN_UNLOCK` in the case of MVAPICH). Consequently, O_1 did not experience Late Unlock and lasted only about $340\mu s$. The lazy lock acquisition approach is immune to Late Unlock because the whole epoch always degenerates to the single unlock call. The consequence, as shown by the O_0 epoch (about $1340\mu s$ for MVAPICH), is a total absence of communication/computation overlapping. The new implementation acquires the lock right away if it is available, and achieves overlapping with both its blocking and nonblocking versions, as shown by the first lock in Fig. 6. The blocking series of the new implementation suffers Late Unlock in the second lock and had to incur the whole duration of the first lock epoch plus its own data transfer latency. In the nonblocking series, Late Unlock is avoided; the O_1 epoch lasts only the duration of the data transfers of O_0 and the one of O_1 itself; but O_1 does not incur the $1000\mu s$ latency created by the work of O_0.

2) Progress Engine Optimizations: All the previous tests were performed with all progress engine optimization disabled. Here, we show the effects of the optimization flags introduced in Section VI-B. The following tests are all performed with nonblocking synchronizations only; but with and without a flag enabled. All the epochs host a single 1 MB PUT; and each subsequent epoch in any given process is opened after the previous one is closed.

A_A_A_R (GATS): We consider a single origin O and two targets T_0 and T_1. The origin opens an access epoch toward T_0 first and then toward T_1. The exposure of T_0 is $1000\mu s$ late, leading to a Late Post situation. The observations of interest are the overall latency of the second target epoch (T_1) and the cumulative origin latency. Figure 7 shows that when `A_A_A_R` is ON, T_1 does not suffer the delay of T_0; and the cumulative origin-side latency is just the latency of T_0. For the origin, `A_A_A_R` allows the second epoch to be overlapped

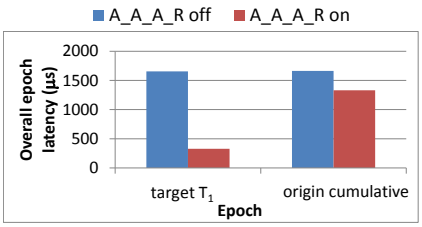


Fig. 7: Out-of-order GATS access epoch progression with A_A_A_R

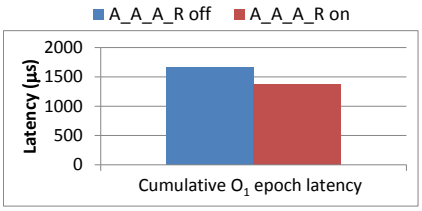


Fig. 8: Out-of-order lock epoch progression with A_A_A_R

with the delay of the first one. When `A_A_A_R` if OFF, the delay of T_0 propagates to the origin, which then propagates it in chain to T_1.

A_A_A_R (lock): We consider two origins O_0 and O_1 and two targets T_0 and T_1. O_1 requests the lock of T_0 right after O_0 gets it. Then O_1 requests a subsequent lock from T_1. Before releasing the first lock, O_0 works for $1000\mu s$ in the epoch. The observation of interest (Fig. 8) is the cumulative duration of both epochs of O_1. When `A_A_A_R` is enabled, O_1 completes both epochs in about $1340\mu s$, which is the latency of its first epoch only, because the second epoch got progressed out-of-order and completed while the first was still being delayed. The delay as well as both epochs is serialized when `A_A_A_R` is disabled.

A_A_E_R: The test setting is made of three processes P_0, P_1, and P_2. P_0 is an origin and P_1 a target. P_2 behaves as a target for P_0 and then as an origin for P_1, in that order. P_0 is $1000\mu s$ late. Figure 9 shows that, by default, the delay of P_0 is transferred to P_2 and then transitively to P_1. When P_2 enables `A_A_E_R`, however, its second epoch, meant for P_1, is progressed and completed out of order. Thus, P_1 completely avoids incurring the delay while P_2 overlaps it with its second epoch.

E_A_E_R: The test setting is made of two origins O_0 and O_1 and a target. O_0 is $1000\mu s$ late. The first exposure of the target is meant for O_0 and the second for O_1. Figure 10 shows that the delay of O_0 is transitively transferred to O_1 by default. The cumulative latency experienced by the target is the sum of that delay and the latency of both its epochs toward O_0 and O_1. With `E_A_E_R` enabled, the delay does not propagate to O_1; and the target also overlaps it with its second epoch.

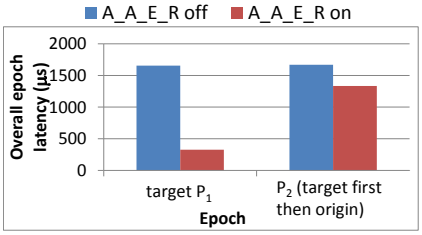


Fig. 9: Out-of-order GATS epoch progression with A_A_E_R

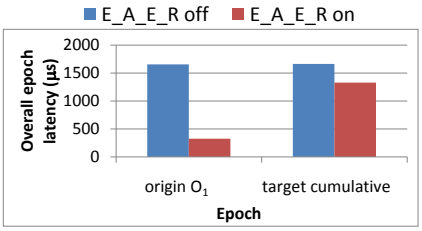


Fig. 10: Out-of-order GATS epoch progression with E_A_E_R

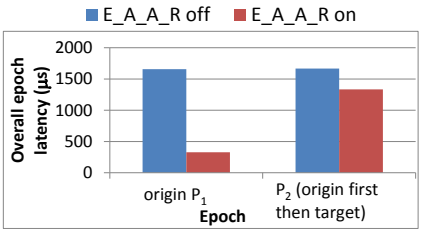


Fig. 11: Out-of-order GATS epoch progression with E_A_A_R

E_A_A_R: The test setting is made of three processes P_0, P_1, and P_2. P_0 is a target and P_1 an origin. P_2 behaves as an origin for P_0 and then as a target for P_1. P_0 is $1000\mu s$ late. Once again, unlike the default case, the activation of `E_A_A_R` by P_2 prevents the propagation of the delay of P_0 to P_1 and allows the second epoch of P_2 to overlap the delay, leading to a lower cumulative latency for P_2 (Fig. 11)

B. Communication Pattern and Application Results

Comparison between the two blocking series (MVAPICH and New) is not the purpose of this work; in fact, for fairness, the nonblocking test series (which is provided by the new implementation) will be compared with the blocking test series of the new implementation. However, the experiments in this subsection show that the New (blocking) series outperforms MVAPICH, sometimes substantially. This observation deserves a brief explanation. Compared with MVAPICH, our new RMA progress engine performs some crucial optimizations. For instance, RMA messages are reordered inside epochs to minimize overall transfer times; and two-sided transfers are avoided altogether, except for compare_and_swap and fetch_and_op. The most important performance difference, however, comes from our strategy of issuing right away the RMA transfers of any target that becomes available. In comparison, if a process opens an access epoch toward a group of 2,048 targets (via `MPI_WIN_START` for instance), MVAPICH waits for all 2,048 targets to open their exposure epochs before issuing any transfer to any of the peers. Consequently, a single late peer can create substantial delays in the MVAPICH test series; and the extent of the delay tends to get worse with the number of peers.

Dynamic Unstructured Massive Transactions: We reproduce in this section the massively unstructured atomic communication pattern described in Section IV-B. An InfiniBand flow control issue prevents the new implementation from scaling beyond 512 processes when there are large numbers of simultaneously pending epochs. This issue is implementation-related and can be fixed, given enough time. The results (Fig. 12) show that the nonblocking version ("New nonblocking") is consistently better than the blocking version ("New"), and the nonblocking with `A_A_A_R` is better than both. The difference between the blocking and nonblocking

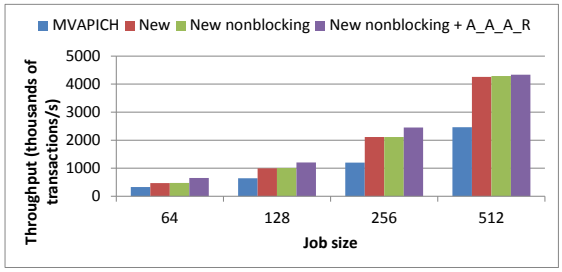
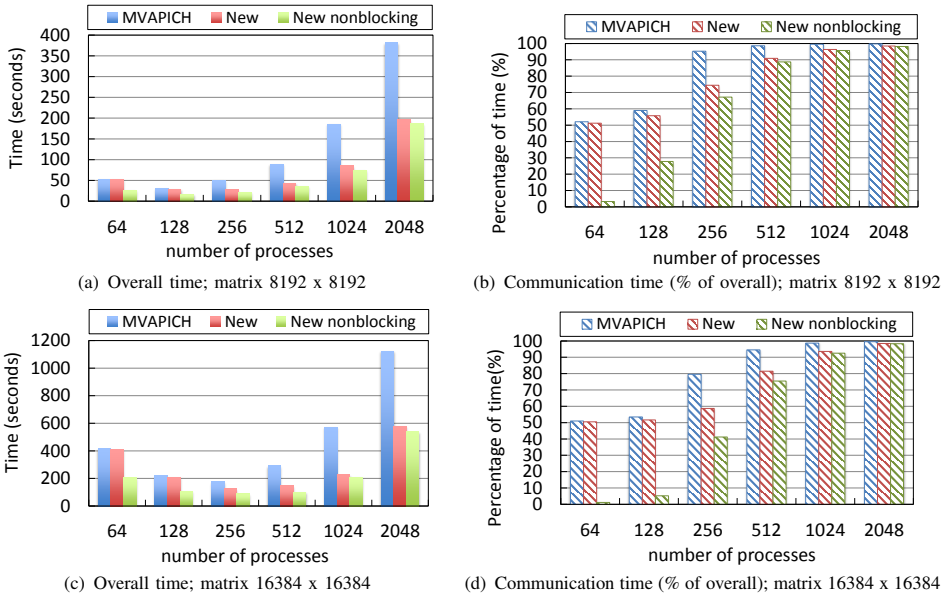


Fig. 12: Massive unstructured atomic transactions

(without `A_A_A_R`) is not noticeable, but it does reach a few thousand transactions/second, That difference is small because the epochs are issued back-to-back and end up being serialized inside the progress engine. That difference would be more substantial if there were computations between adjacent transactions. Rather than communication/computation overlapping, the improvement opportunities of this communication pattern come mostly from contention avoidance, as enabled by `A_A_A_R`. `A_A_A_R` allows 184,422 (39%), 205,377 (20%), and 339,359 (16%) more transactions per second than does the blocking test series in jobs of 64, 128, and 256 CPU cores, respectively. Because of the aforementioned flow control issue, that difference is only 47,263 transactions per second (2%) with 512 CPU cores.

LU Decomposition: Figure 13 presents the performance results of a lower-upper (LU) decomposition for solving square systems of linear equations. We implemented a kernel of 1D LU decomposition by using GATS epochs. The algorithm does cyclic mapping to ensure load balance and concurrency. For a matrix of size $m \times m$ and for a job size n, each process gets m/n matrix rows. Then when a row (in upper triangle) belonging to a process P gets updated, P broadcasts its nonzero cells (one-sidedly) to the other $n-1$ peers. At fixed m, when n grows, each process gets fewer and fewer m/n rows to broadcast to larger and larger numbers $(n-1)$ of peers; that is, each process experiences fewer but heavier communications. At fixed matrix size, these two conflicting effects of n have the consequence of decreasing the overall execution time of LU up to a certain optimal job size and then increasing it from there on. In Fig. 13(a) for matrices of $8k \times 8k$, with the job increase steps used, the optimal job size is 128 processes. In Fig. 13(c) for matrices of $16k \times 16k$, that optimal size is 256 processes. Nevertheless, the tests are executed up to 2048 processes because the observation of interest is simply how the "New nonblocking" test series compares with the "New" test series for each job size.

The program has two kinds of communication/computation overlapping: inside the epoch (exists in all three series) and after the epoch is closed but not necessarily completed (exists only in "New nonblocking"). The program offers considerable room for the first kind of overlapping. In the blocking version ("New"), however, that overlapping leads to the Late Complete issue. The "New nonblocking" test not only eliminates the Late Complete issue but also enables the second kind of overlapping without generating any inefficiency, leading to performance improvements of 50% (64 to 128 processes in Fig. 13(a) and Fig. 13(c)). Figure 13(b) and Figure 13(d) show the percentage of the overall execution times that the CPUs spend in MPI communication calls. At fixed matrix size, one can see that when the job size increases, the communication percentages



(a) Overall time; matrix 8192 x 8192

(b) Communication time (% of overall); matrix 8192 x 8192

(c) Overall time; matrix 16384 x 16384

(d) Communication time (% of overall); matrix 16384 x 16384

Fig. 13: Performance evaluation by LU decomposition

increase because of the decreased amount of computation per process. This behavior leads to the shrinking of the extent of the Late Complete issue and consequently justifies the shrinking of the advantage provided by the "New nonblocking" test series in Fig. 13(a) and Fig. 13(c) when job sizes grow.

IX. Related Work

The MPI specification offers considerable leeway to implementers about when to force the waits in an epoch. In particular, an access epoch opening synchronization call does not have to block if the corresponding exposure epoch is not opened yet. This freedom is used in [8], [9] to mitigate the apparent effects of RMA synchronization by deferring the actual internal execution of both synchronization and communication to the epoch-closing routine execution. This approach is termed *lazy*. In [13] a design of the fence epoch is proposed where communication/computation overlapping occurs inside the epoch. Since the proposal makes every single fence call blocking, however, the overlapping comes at the price of a potentially substantial idleness at both opening and closing of each epoch. Purely intranode RMA issues have been addressed in [14], [15]. An approach to the hybrid design of MPI one-sided communication on multicore systems over InfiniBand is presented in [16]. The work described a way of migrating passive target locks between network atomic operations and CPU-based atomic operations. The use of RDMA for one-sided communications was presented in [17], [18], [19]. Designs of the computational aspects of `MPI_ACCUMULATE` were proposed in [20], [21]. In [12], a strategy is proposed to adaptively switch between lazy and eager modes for RMA communications to achieve overlapping. An MPI-3.0 RMA library implementation for Cray Gemini and Aries systems is described in [22]. To the best of our knowledge, however, none of the previous work altered the epoch-closing routines of MPI RMA to render the one-sided communications lifetime nonblocking from start to finish. Thus, the work presented here pioneers entirely nonblocking MPI one-sided synchronizations proposals and designs.

X. Conclusion and Future Work

The blocking nature of MPI one-sided epoch-closing synchronizations can lead to latency propagation toward peers linked in matching epochs. These issues are documented and categorized in six inefficiency patterns of which four could not effectively be worked around. We introduce in this work a new, previously undocumented inefficiency pattern. Then we propose entirely nonblocking RMA synchronizations. We show that all the four unaddressed inefficiency patterns as well as the newly documented one are now solved with our proposed one-sided communication synchronizations. The nonblocking synchronizations lead to an increased potential for communication/computation overlapping as well as delay mitigation via communication/delay overlapping. The nonblocking epochs also allow new use cases of HPC communications, such as those that require multiple epochs to be issued back to back, in order to be more efficiently handled. Since nonblocking synchronizations and epochs bring additional complexities, we spell out their semantics, the hazardous situations and the behaviors to expect from the progress engine.

As future work, we intend to investigate the possibility of enabling the progress engine optimization flags for fence epochs. We are also interested in observing the behavior of the nonblocking synchronizations on petascale-level machines.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada Grant #RGPIN/238964-2011, Canada Foundation for Innovation and Ontario Innovation Trust Grant #7154. We thank Mellanox Technologies for the resources. This work was also supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contracts DE-AC02-06CH11357, DE-FG02-08ER25835, and DE-SC0004131. We thank Argonne National Laboratory for use of the LCRC Fusion cluster.

REFERENCES

[1] MPI Forum, "The Message Passing Interface," http://www.mpi-forum.org/, online; accessed 2013-12-04.

[2] D. Bonachea and J. Duell, "Problems with Using MPI 1.1 and 2.0 as Compilation Targets for Parallel Language Implementations," *International Journal of High Performance Computing and Networking (IJHPCN)*, vol. 1, no. 1-3, pp. 91–99, Aug. 2004. [Online]. Available: http://dx.doi.org/10.1504/IJHPCN.2004.007569

[3] A. Kühnal, M.-A. Hermanns, B. Mohr, and F. Wolf, "Specification of Inefficiency Patterns for MPI-2 One-Sided Communication," in *Proceedings of the 2006 Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 4128. Springer, August - September 2006, pp. 47–62.

[4] M.-A. Hermanns, M. Geimer, B. Mohr, and F. Wolf, "Scalable Detection of MPI-2 Remote Memory Access Inefficiency Patterns," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5759, pp. 31–41. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_10

[5] "MPICH: High Performance Portable MPI," http://www.mpich.org/, online; accessed 2013-12-11.

[6] "MVAPICH," http://mvapich.cse.ohio-state.edu/, online; accessed 2013-12-11.

[7] "Open MPI," http://www.open-mpi.org/, online; accessed 2013-12-11.

[8] B. Barrett, G. Shipman, and A. Lumsdaine, "Analysis of Implementation Options for MPI-2 One-Sided," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, vol. 4757, pp. 242–250. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75416-9_35

[9] R. Thakur, W. Gropp, and B. Toonen, " Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication," *International Journal of High Performance Computing Application (IJHPCA)*, vol. 19, pp. 119–128, 2005.

[10] "OpenFabrics Alliance," http://www.openfabrics.org/, online; accessed 2013-12-13.

[11] J. A. Zounmevo and A. Afsahi, "A Fast and Resource-Conscious MPI Qessage Queue Mechanism for Large-Scale Jobs," *Journal of Future Generation Computer Systems*, vol. 30, no. 0, pp. 265–290, 2014.

[12] X. Zhao, G. Santhanaraman, and W. Gropp, "Adaptive Strategy for One-Sided Communication in MPICH2," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7490, pp. 16–26. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33518-1_7

[13] G. Santhanaraman, T. Gangadharappa, S. Narravula, A. Mamidala, and D. Panda, "Design Alternatives for Implementing Fence Synchronization in MPI-2 One-sided Communication for InfiniBand Clusters," in *Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops (Cluster)*, 2009, pp. 1–9.

[14] P. Lai, S. Sur, and D. Panda, "Designing Truly One-sided MPI-2 RMA Intra-node Communication on Multi-core Systems," *Computer Science - Research and Development*, vol. 25, no. 1-2, pp. 3–14, 2010. [Online]. Available: http://dx.doi.org/10.1007/s00450-010-0115-3

[15] M. Luo, S. Potluri, P. Lai, E. Mancini, H. Subramoni, K. Kandalla, S. Sur, and D. Panda, "High Performance Design and Implementation of Nemesis Communication Layer for Two-Sided and One-Sided MPI Semantics in MVAPICH2," in *Proceedings of the 2010 International Conference on Parallel Processing Workshops (ICPPW)*, 2010, pp. 377–386.

[16] G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp, and D. K. Panda, "Natively Supporting True One-Sided Communication in MPI on Multi-core Systems with InfiniBand," in *Proceedings of the 2009 IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE Computer Society, 2009, pp. 380–387. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2009.85

[17] W. Jiang, J. Liu, H.-W. Jin, D. Panda, W. Gropp, and R. Thakur, "High Performance MPI-2 One-Sided Communication over InfiniBand," in *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2004, pp. 531–538.

[18] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen, "Design and Implementation of MPICH2 over InfiniBand with RDMA Support," in *Proceedings of the 2004 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2004.

[19] G. Santhanaraman, S. Narravula, and D. Panda, "Designing Passive Synchronization for MPI-2 One-sided Communication to Maximize Overlap," in *Proceedings of the 2008 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2008, pp. 1–11.

[20] W. Jiang, J. Liu, H.-W. Jin, D. Panda, D. Buntinas, R. Thakur, and W. Gropp, "Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, vol. 3241, pp. 68–76. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30218-6_16

[21] J. Nieplocha, V. Tipparaju, and E. Apra, "An Evaluation of Two Implementation Strategies for Optimizing One-Sided Atomic Reduction," in *Proceedings of the 2005 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005, pp. 215.2–.

[22] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided," in *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2013, pp. 53:1–53:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503286